RAIINET

By Dalya Tasli, Laeticia Fosso, Andrea Ma

Introduction

Through collaborative efforts, our developers Dalya Tasli, Laeticia Fosso, and Andrea Ma have completed our journey of implementing this game. In this document, we will take you through the most major aspects of our code, talk about how we ensure resilience to change, and talk about how our answers to the specified questions have changed since starting to implement our code.

Overview

The first aspect of our code we want to mention is our use of design patterns. We implemented the observer design pattern with our notify class being the subject, and our text and graphics classes being the observers. The notify class has all of the methods that are needed in the subject, such as attach, detach as well as notify all observers. We keep track of all of our observers in a vector which is how we are able to notify them all efficiently. The observers get notified when either player types in the command line “move”. After they move any link of their choice, then that link’s status should update in the Link class, as well as the cell it is now on should update to state that it is now being occupied, then the text and graphics classes get notified. The change that occurs in our bobservers when they are notified is simply that they display a new graphic. This is why the most important aspect of our observer class is to make sure that our graphics and text are being updated.

Another aspect of our code that is important are our link and cell classes. The cell and link classes work very uniformly, as they both have to react for the notify all observers method to be called. The cell class deals with each board cell, which is needed when working with abilities such as firewall, to check if a particular box is free, to see which link (if any) is currently on it, and to get its coordinates. Similarly, the link class is used to get the actual data of each link, such as data or virus, the player ID, and

its strengths. After we use the methods from both of these classes, then the text and graphic display can observe them and display the changes made.

Another very important aspect of our code is the player class. This is the class that deals with which player we are currently on, as well as actually creating the players. This class keeps track of all of the information about each player. The way we implemented this is through having 2 player fields that store all of the information about these players. This lets us access the stats of each player. We decided that it would be the most efficient to keep all of the information about the abilities in our player class, to then call in our main function but still be stored all in one place.

Initially, we wanted to implement our code in a way that every class only did one thing by the MCV method, which meant that our main function would be reading in only one set of input for the initialization of the game, then call a different class to get the command line input for the controlling of the game. However after careful consideration, we changed this so that all of the input is read in main as we found this to have the least memory leak errors as well as to keep a strong encapsulation. Otherwise, we would have to have a lot of public fields and methods, and we wanted to keep as many private fields as possible, with the least number of friends.

Design

The first design challenge we had was figuring out how to implement our text display cohesively with the rest of our code's implementation. This is because we had to remember the current position of a link before it moved, then move it to the new position, but still remember the old position in order to be able to change its state in the text display. In order to overcome this challenge, we used a visualization and step-by-step analysis technique. Our team drew out a rough sketch of the board on a white board and made step by step notes of everything that had to happen in oder for us to be able to implement the command line interactions, such as the abilities and the move function. After visualizing the board and writing out our steps, we were able to come up with the solution of making a separate cell class in order to be able to remember the old coordinates, and let the positions on the graph have their own fields that can remember their states and their values, instead of only storing the current cell the links are on in our link class. Had we not used our technique of visualizing the board as a team, we would not be able to come up with the idea of implementing a

separate cell class. Our solution enabled us to have good organization and maintenance, as well as helped us realize the importance of teamwork and doing step-by-step analysis as a team.

The next design challenge we had was implementing the download function. This is because we first only thought of download as the ability, and when we finished implementing the ability download, we thought we would make copies of it so we could use it for downloading when on the other side of the board, or downloading after a battle. However, this became very challenging to implement, and to keep track of multiple download functions with different parameters that did different jobs. This is why we used the technique of parametrized functions combined with adaptability and team collaboration. As a team, we decided to each come up with the different jobs per download function, then all told each other what our own specific download function had to complete, similar to each expertizing on a single download function. This way, we were able to find similarities in our code and come up with merged parameters. After coming up with one big download function with multiple parameters, we now had to make a default variable of each parameter that was not called every single time. For example, for downloading as an ability, you do not need the current player as a parameter because the main function takes care of that, however when downloading because we reached the other side of the board, we needed to specify the player which was downloading each link as different players get to download the same link depending on whether or not they are on a server port or not. This lead us to making our player parameter default, so we did not have to give in a player if it was an ability. This technique helped us to practise adapting functions and streamline our code, as well as organized and helped keep our code efficient.

Resilience to change

In order to support the possibilities of changes to program specification, we kept all of the information or states as fields that can be modified easily in only one place, with only main.cc changing the control flow. This way, if we get any changes that we need to implement, we would simply have to take in more input in main, and make quick and easy changes to our fields in player class. We ended up adding a cell class that was not on our initial UML diagram as we thought it was a good place to keep data

about each cell so that if there are new changes to be made, they can be made through fields in this class instead of making changes to methods by passing by references.

A big way we encouraged our resiliency to change is through making sure we had good encapsulation at the same time, as well as high cohesion. In our code, we have high cohesion. All of our classes and functions cooperate together to do basic tasks, with a few elements that pass each other data. This was the most important way we made sure that any changes could be implemented with low effort. By making sure we have well organized code that is easy to maintain and understand, we were able to ensure high cohesion. On the other hand, we have limited coupling. Most of our interactions to different classes and modules happen through function calls and passing in parameters. We also made sure to not have friends or global data in order to have good encapsulation and low cohesion.

Answers to questions

1) In order to have two displays of both views, we would first of all need to change our main function. In main, we switch between players to take in the input and print the display after we move. We would have to change the move function in our main so that instead of switching the display, we just update it. Next, we would have to greatly change our graphics display. Instead of keeping track of two separate displays and switching, we would have to shrink our implementation and fit them both on one screen. This differs from our first answer, as we did not think that it would affect our main function as much as it did. Main is what controls all of the outputting including the graphics, so we would definitely have to take out our toggle function and not keep displaying different graphics.
2) To make adding new abilities easy, we can make them into methods in our link class. This way, we don't need to create any new interface files for new classes or files, we would also have access to all of the fields in the link class. Since most of the abilities have to do with changing the fields, then it would be the easiest to implement them all and just add them straight into Link. For due date 1, we planned this out well so we do not have any differences in our answers.
3) In order to make these changes, similar to the previous question, we would have to most importantly change our graphics display in order to fit all four quadrants. Next, we would have to change our main function and extend our

players array to four players instead of two. This would then change a handful of other implementations. First of all, the initialization of our game would be different. We would have to take in the links and abilities for all four players instead of two. We would then have to change the game command input portion of our main function in order to change the correct links, or run the right abilities for the right players. Next in our board class, we would have to extend the function that checks for valid position to move our link, since we would now be able to go to the left or right quadrants as well. We would also have to change our text display in order to output for four players instead of two. We don't think that it would only require minimal changes to have this implemented in addition to our two player mode, as the graphics and our variables that keep track of both of our players' information would have to be completely changed and big additions would have to be made. This is very different from our answer in DD1, as we thought this would be a lot more simple to implement before we started the actual implementation of our code, and we realize now that there are a lot of tiny details that would have to be tweaked in order to add this change.

Extra credit functions

Final Questions

1) Some important lessons we learned about developing software in teams is that different people think in different stages of the coding process. For example, if one person starts off the code by writing all of the interface files then filling them in on the implementation file method by method. On the other hand, another group member might start with the implementation file and add the method names on the interface one by one as each implementation is completed. This can create some problems as at a set time, it might look like one has done more work than the other, purely because of the difference in the number of files implemented, or the number of codes written. We have also learned that it is important to set some basic base points on how to implement specific logistics. For example, we can all agree to make use of while loops more often than for loops, or use if and else statements rather than recursion in order

to keep things flowing and looking the same. Most importantly, communication and commenting is very important. It is easy to forget that not everyone can read the code you wrote and understand it exactly the way you meant for it to be, or understand what all of the fields are for. By using lots of comments, it reduces the confusion and speeds up the process. Also for communication, it is important to keep each other updated on which file you are working on, so that others don't work on it at the same time as you. This can lead to the code being overwritten and lost.

2) If we had a chance to start over, we would start a UML before we started, and fit the code to the UML instead of making the UML to fit the code. This would have made it easier for everyone and especially to break up the work. If we all spent more time gathering ideas and implementing our UML diagram, then it would have sped up our individual coding. This is because we would all be on the same page for each method to be implemented in each class, and we wouldn't have to wait for each other to respond in our group chats to any questions one might have about someone else's class or methods. It would also speed up the process as it could really help assign a file to a different person so they could all work on different things at the same time. Another thing we would have done differently is start the main file at the end. It made things harder to try to fit code to the main file instead of fitting the main file to the code.

Conclusions

In conclusion, our team's significant efforts in the last two weeks has been summarized in this document. We hope you enjoy our game and are in a good mood when marking it :)